<알고리즘 요약 정리본>

3_2장 고급 정렬 (170327 to 170403)

우리가 여태껏 배웠던 기본 정렬에서 대부분 시간 복잡도가 O(n^2)이었다. 하지만 고급 정렬에 대한 대부분 시간 복잡도가 O(n*log n)이 나오는 기적의 상황이 나올 수도 있다. 물론 병합 정렬, 퀵 정렬 보다 시간 복잡도가 더 낮은 기수 정렬(O(n))도 존재하지만, 우리가 배우는 수준인 병합 정렬, 퀵 정렬, 힙 정렬에 대해서도 접근하는데 있어서 큰 어려움이 있으니 강의 노트를 제대로 읽어봐야 할 필요가 있다.

1. 병합 정렬(Merge Sort)

우선 병합 정렬은 예를 들어 아래와 같은 배열이 정의되어있다고 가정하자.

10	8	22	4	9	14	20	3

과거에 생물 시간에 세포 분열에 대해서 공부를 해봤을 것이다. 이 원리를 이용해서 위 배열에 대하여 하나의 데이터를 분할시켜 보면 (10), (8), (22), (4), (9), (14), (20), (3) 8개의 값이 나올 수 있다. 하지만 분할을 했다면 어떻게 해야 되는가? 당연히 결합을 해서 정렬을 할 필요가 있다. 10은 8이랑, 22는 4랑, 9는 14랑, 20은 3이랑 비교해서 (8, 10), (4, 22), (9, 14), (3, 20) 이렇게 나온다. 이러한 원리로 (4, 8, 10, 22), (3, 9, 14, 20) 이렇게 뭉쳐서 결국에는 (3, 4, 8, 9, 10, 14, 20, 22) 이처럼 뭉치게 되겠다. 이러한 원리가 처음에 분열되어 있다가 뭉치게 되는 병합의 원리를 적용한 병합 정렬이다. 병합 정렬에 대해서 어떻게 돌아가는지에 대해서 이번에는 대략적인 그림이 아니라 코드를 분석하면서 알아보도록 하자.

2. 병합 정렬 알고리즘과 원리

```
public static void mergeSort(int[] arr,
int start, int end){
    if(start<end){
      int middle=(start+end)/2; // 0
      mergeSort(arr, start, middle); // 1
      mergeSort(arr, middle+1, end); // 2
      merge(arr, start, middle, end); // 3
    }
}
```

0 : 우선 배열에 대해서 나누어서 생각하기 위해서는 middle(중간 값)을 고려해야 된다. 그래서 시작점과 끝점 사이에 대해 중간 값을 계산해서 이용하면 되겠다.

1 : 이는 middle을 기준으로 나뉘었을 경우에 처음 배열부터 중간 인덱스까지 앞부분에 대해서 분할을 해서 병합 정렬을 하는 과정으로 살펴보면 되겠다.

2 : 중간 인덱스 이후로부터 마지막 인덱스까지 나뉘었을 경우에 뒷부분에 대해서 병합 정렬을 하는 과정으로 살펴보면 되겠다.

3 : 실질적으로 병합 정렬이 이루어지는 과정은 이 부분에 있는 merge 알고리즘에서부터 시작이 된다. 이에 대해서는 작년 C언어 경진대회에서 잠깐 다룬 기억이 있을 것이다. 이를 이용해서 어떠한 알고리즘이 이루어지는지에 대해서는 위에서 다시 설명하겠다.

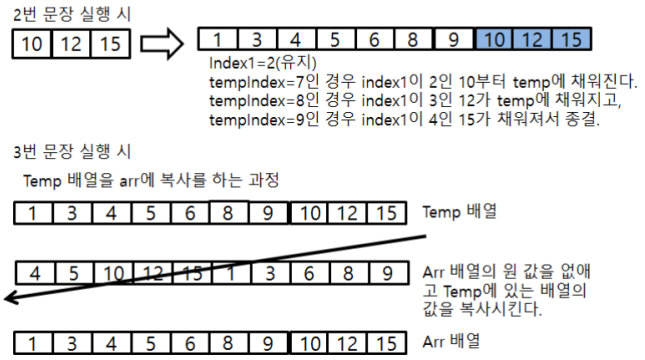
1번과 2번은 재귀적 호출로 인해서 배열의 크기를 n으로 가정을 하면 $\log_2 n \times 2$만큼 배열을 중간 값을 기준으로 분할하는 작업을 한다. 그리고 merge 메소드에서 실행되는 횟수는 n이 되시겠다. 그래서 병합 정렬의 시간 복잡도는 $O(n \log n)$이 된다. 이제 그 문제의 merge 메소드를 살펴보도록 하자.

```
public static void merge(int[] arr, int start,
int middle, int end){
  int index1=start;
  int index2=middle+1; // 0
  int length=end-start+1;
  int[] temp=new int[length];
  int tempIndex=0;
  while(index1<=middle && index2<=end){
    if(arr[index1]<arr[index2])
      temp[tempIndex++]=arr[index1++];
    else
      temp[tempIndex++]=arr[index2++]; // 1
  while(index1<=middle){
    temp[tempIndex++]=arr[index1++];
  }
  while(index2<=end){
    temp[tempIndex++]=arr[index2++];
  } // 2
  for(int k=0;k<temp.length;k++){
    arr[start+k]=temp[k];
  } // 3
}
```

이 merge라는 알고리즘은 예를 들어 (4, 5, 10, 12, 15) / (1, 3, 6, 8, 9) 이라는 정렬된 배열에 대해서 살펴보도록 하자.

1 : 쉽게 접근해본다면, 우리가 어렸을 때 유희왕 카드 게임을 해봤을 것이다. 예를 들어 내가 공격력이 3000정도 되는 카드를 내밀 경우에 상대방이 공격력이 5000정도 되는 카드를 내밀면 그 카드는 게임에서

못 쓰게 된다. 이렇게 못 쓰는 카드들을 내려놓다보면 최소한 공격력이 낮은 카드부터 계속 정리될 것이다. 이러한 원리가 병합 정렬의 원리로 생각하면 되겠다. 만일 마지막에 어느 선수 중에 분명 카드를 다 쓰는 경우가 존재할 것이다. 이러한 경우에는... 2번과 같은 문장을 참조하면 되겠다.



2 : 이는 index1, index2 중에서 나머지에 대해서 정리를 하는 것이다. 게임을 시작하면 무승부라는 개념이 없다면 둘 중 하나는 패배를 하게 되듯이, 남는 것들에 대해서 temp 배열에 완성을 해주는 과정으로 생각하면 되겠다.

3 : 마지막으로 temp 배열을 arr 배열에 복사를 해주는 과정으로 살펴볼 수 있다. 그렇지만 여기서는 start가 0일 때를 가정을 해서 작성을 해왔지만, 실제로는 배열을 쪼개는 경우에 start가 0이 될 수도 있고, 2가 될 수도 있고 그건 모르는 일이다. 여하튼간에 강의 노트를 참조하게 되면 start~end 범위 내에서 병합 정렬이 끝난 부분을 제외하면 나머지는 안 된 부분이 있지만 대부분 오른쪽 부분이다. 그리고 반반씩 정렬이 모두 완료된다면 그림으로 설명한 방식으로 정렬이 진행되는 것을 확인할 수 있다.

<참조> 병합 정렬보단 퀵 정렬을 더 선호하는 이유??
병합 정렬과 퀵 정렬의 수행 시간은 O(n logn)이다. 하지만 병합 정렬은 각 배열에 있는 요소들을 하나씩 분할을 해야 되기 때문에 공간 복잡도가 O(n)이 된다. 그렇지만 퀵 정렬은 pivot라는 녀석을 이용해서 정렬을 마치기 때문에 공간 복잡도에 대해서는 고려를 안 해도 된다. 하지만 병합 정렬에 비해서 매우 난해한 알고리즘이라서 이에 대해 이해를 하려면 실제로 그림을 그려가면서 원리를 깨치는 방안 이외에는 없다... ㅜㅜ

3. 퀵 정렬(Quick Sort)

알고리즘 공부를 떠나서 잠깐 노래 이야기를 꺼내보겠다. 한 2000년대 후반 쯤 래퍼 Dok2, Double K가 '훔쳐' 라는 노래를 불렀다. 그렇지만 노래에 뭔가 재미가 없어져서 2010년 초반에 Double K가 자기 부르는 부분에 대해서 속사포로 랩을 해버리는 형식으로 최근에 라이브를 뛰고 있다. 이처럼 퀵 정렬도 같은 데이터에 대해서 일반 정렬(O(n^2)의 시간 복잡도)에 비해 빠른 시간 복잡도인 O(n log n)으로 끝내면서, 공간 복잡도가 O(n)인 병합 정렬과 달리 pivot라는 개념을 이용해서 공간 복잡도를 O(1)로 줄이면서 효율적인 정렬을 보여주는 알고리즘이 바로 퀵 정렬이다. 그래서 퀵 정렬에 대해 대략적으로 어떤 원리로 돌아가는지에 대해 숙지를 하고 넘어갈 필요가 있다.

3-1) 퀵 정렬의 대략적인 구조

쉽게 접근을 해보는 방안에 대해서 설명하면 대략 8명이서 가위 바위 보를 하면서 순위를 결정한다고 가정을 해보자. 병합 정렬을 사례로 설명하면 2명이서 끼리끼리 가위 바위 보를 해서 토너먼트로 진행하는 것이 방안인데, 퀵 정렬은 몇 명으로 나뉘지 않고 바로 가위 바위 보를 해서 이긴 사람끼리, 진사람 끼리 가위 바위 보를 해서 순위를 짜는 방안이 바로 퀵 정렬이다. 이처럼 어떤 기준에 대해서 값들을 왼쪽으로 보내고, 오른쪽으로 보내는 방법이 바로 퀵 정렬에 대한 대략적인 원리이다.

아래의 배열의 요소처럼 이를 참조해서 설명에 들어가보자.

10	8	22	4	3	7	20	14

음영 부분을 둔 값이 바로 기준 값이다. 그니깐 이를 이용하면 아래와 같이 분류가 되겠다.

10	8	4	3	7	14	22	20

14보다 작은 값들을 모은 왼쪽을 1구역이라고 생각하면 되고, 반대로 큰 값이 있는 오른쪽을 2구역으로 생각해볼 수 있다. 여기서 또 1구역과 2구역을 기준 값을 구역의 마지막 값을 통해서 다시 그 구역 내에서 옮기는 원리를 생각해보자.

4	3	7	10	8	14	22	20

이처럼 7를 기준으로 값이 다시 잡혔다. 하지만 아직까지도 정렬이 제대로 이루어지지 않았기 때문에 다시 정상적으로 정렬을 완료하기 위해 어두운 음영에 대해서 정렬 작업을 완료시켜보면 아래와 같은 과정으로 정렬을 완료할 수 있다.

4	3	7	10	8

(정렬 前)

3	4	7	10	8

(1구역 내의 1구역 정렬 완료)

3	4	7	8	10

(정렬 後. 1구역 내의 2구역 정렬 완료)

이처럼 정렬을 완료하게 된다면 출력을 하는 것으로 종결을 낸다. 그리고 처음에 지정된 2구역에 대해서도 다시 정렬 작업에 들어가게 되면 아래처럼 정렬이 완료가 되는 사실을 알 수 있다.

3	4	7	8	10	14	20	22

3	4	7	8	10	14	20	22

이처럼 기준 값을 이용해 작은 값들은 1구역으로, 큰 값들은 2구역으로 보내는 원리가 바로 퀵 정렬이다. 말만 들어도 쉽게 느껴지겠지만, 이 알고리즘은 생각보다 꽤 난해한 알고리즘이다. 어떤 알고리즘으로 작성이 되어있는지에 대해서 정확히 분석하면서 이런 원리로 돌아가겠다는 사실에 대해 공부를 하면서 이해를 해보는 것으로 요약정리를 작성해 보겠다.

3-2) 퀵 정렬 알고리즘

퀵 정렬 알고리즘에는 partition 메소드랑 quickSort 메소드로 2가지로 나뉘게 된다. 여기서 quickSort 메소드는 mergeSort 메소드랑 뭔가 유사하다는 점을 발견할 수 있을 것이다. 그런데 강의 노트에 있는

middle은 중간 값이라고 이해하는 혼돈이 이루어질 수 있으니 pivot로 재작성해서 정리하겠다.

```
public static void quickSort(int[] arr,
int start, int end){
    if(start>=end) return;
    int pivot=position(arr, start, end); // 1
    quickSort(arr, start, pivot-1);
    quickSort(arr, pivot+1, end); // 2
}
```

마치 mergeSort(int[] arr, int start, int end) 메소드와 비스무리하게 느껴질 수 있지만, 여기서 차이점을 짚고 넘어가야 하는 점에 대해서 세부적으로 정리해보자.

1 : mergeSort 일부에서는 아래와 같이 merge라는 메소드를 이용해서 정렬 작업에 들어갔다. 하지만 quickSort는 position이란 메소드를 통해서 기준점을 잡아서 작은 값은 왼쪽, 큰 값은 오른쪽에 두는 원리로 작동을 한다.

```
mergeSort(a[], start, end){
    ...
    merge(a[], start, middle, end);
}
```

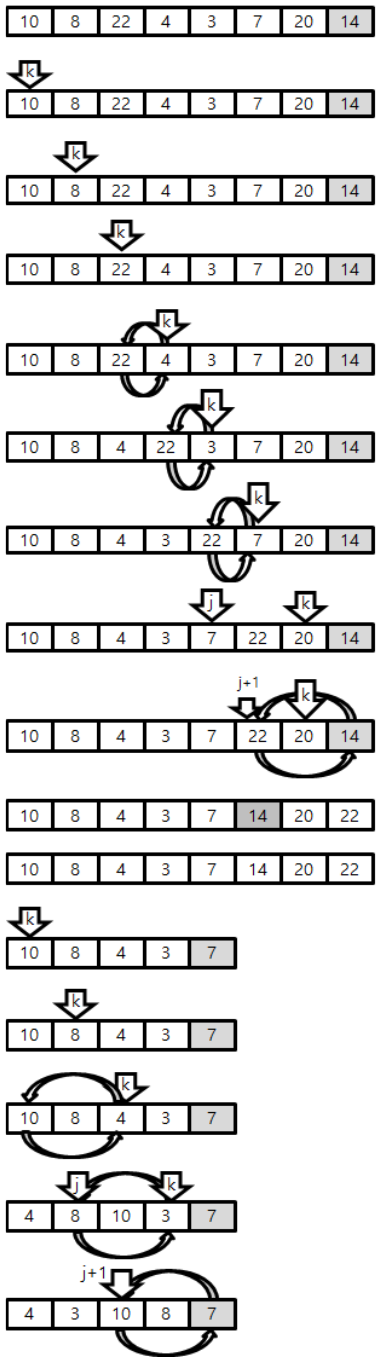
이 메소드에 대해서 다시 생각을 해보겠다면, 여기서는 각 데이터에 대해서 정렬 작업까지 다 해주지만, temp 배열에 인해서 공간 복잡도가 O(n)이 되어버리는 바람에... position 이란 메소드는 quickSort 메소드 분석이 끝나고 다시 공부해보겠다.

2 : quickSort() 메소드는 mergeSort() 메소드와 마찬가지로 $2*\log n$번의 회수만큼 반복된다는 사실을 알 수 있다. 다만 mergeSort와의 차이는 mergeSort는 start에서 middle-1끼리, middle에서 end까지 나뉘어서 하지만, quickSort() 메소드는 start에서 pivot-1끼리, pivot+1에서 end까지 나뉘어서 $\log n$번 반복하게 된다.

이처럼 quickSort 메소드를 돌게 된다면 처음에는 log n번 재귀 호출을 하게 되고, partition 메소드 내에서 작동되는 시간 복잡도가 O(n)이 되기 때문에 퀵 정렬의 시간 복잡도는 $O(nlogn)$이 된다. 이제 여기서부터 좀 난해하게 느낄 것이다. 바로 partition 메소드에 대해서 자세하게 공부를 해보겠다.

```
public static void swap(int[] arr, int x, int y){
    int temp=arr[x];
    arr[x]=arr[y];
    arr[y]=temp;
}
public static int partition(int[] arr, int start,
int end){
    int value=arr[end];
    int j=start-1;
    for(int k=start;k<=end-1;k++){
      if(arr[k]<=value)
        swap(arr, k, ++j);
    swap(arr, j+1, end);
    return j+1;
}
```

어우... 생각만해도 복잡하다. 그래서 아까와 같은 배열을 통해서 그림을 이용해 심층적으로 들어가보자.



이러한 원리로 마지막에 quickSort(arr, 6, 7)를 실행하게 되면 아래와 같이 정상적으로 정렬된다.

3	4	7	8	10	14	20	22

(추가) 퀵 정렬에 대해 그림이 작으니 추가로 그림을 다시 보내겠다. 작게 보이면 이를 참고하는 것이 좋겠다.

3-3) 퀵 정렬 수행 시간
퀵 정렬의 일반적인 수행 시간은 $O(nlogn)$이다. 아시다시피 병합 정렬과 같은 원리로서 log n에 비례해서 재귀 함수를 반복하게 되는 것으로 반복이 된다. 하지만 병합 정렬과는 달리 메모리 공간은 O(1)이기 때문에 최선의 경우와 최악의 상황에 대해서 나뉘게 된다.

최선의 경우 : 아시다시피 기준점에 대해서 1구역의 데이터 개수들과 2구역의 데이터 개수들이 같은 경우에 수행 시간에는 $O(nlogn)$이 되겠다.

최악의 경우 : partition 메소드를 이용하여 기준값만 낙오되는 경우에는... 재귀 호출 횟수는 O(n)이 되어서 O(n^2)이 될 수도 있으니... 예를 들어 아래와 같이 올바르게 정렬된 데이터들에 대해서는 최악의 상황으로 살펴볼 수 있다. 기준 값이 계속 최댓값(혹은 최솟값)이 되는 경우에 대해서는 정렬하는데 있어서 무의미해지게끔 낭비가 되기 때문에 실질적으로 퀵 정렬에 대해서는 큰 보장을 못하니 이에 대해서는 융통성있게 알아서 쓰면 좋겠다.
예시> 10, 12, 14, 16, 18, 20, 22...

<참조> 마지막 값을 비교하는 것이 아니라 처음 값을 기준 값으로 결정하는 경우에는 아래와 같은 메소드로 완성이 가능하다. end에서부터 거꾸로 비교를 해서 compare보다 크기가 큰 경우에는 swap을 해주는 원리로 작성할 수 있겠다.

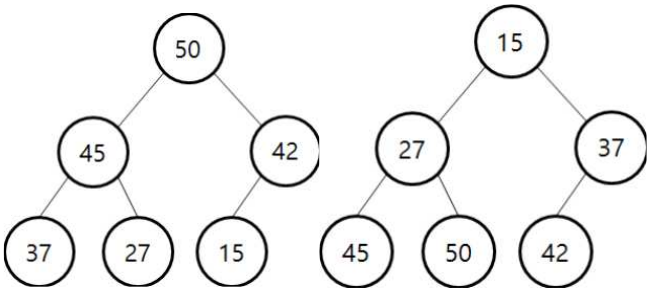
```
public static int partition(int[] arr, int start, int end){
    int compare=arr[start];
    int j=end+1;
    for(int k=end;k>=start+1;k--){
        if(arr[k]>compare){
            swap(arr, k, --j);
        }
    }
    swap(arr, start, j-1);
    return j-1;
}
```

4. 힙(Heap) (참고용)

힙 정렬은 사실 강의노트에 나와 있지 않아서 그렇게 중요하게 다루지 않은 점에 대해 아쉬워서 알아두면 약이 되는 개념이기 때문에 여기서는 힙 정렬 알고리즘에 대해서는 **자료구조에서 배웠던 개념들을 통해서만 간략 설명**으로 하고 넘어가도록 하겠다.

우리가 자바나 C++를 공부한 경우에 객체에 대해서는 참조 값에 대해서 Stack 저장 공간에 저장이 되지만 그 참조 값이 가리키는 객체에 대해서는 실질적으로 Heap 저장 공간에 저장이 된다. 객체가 생기는 원리에 대해서는 동적 할당을 받은 경우에 만들어지는 것이 원칙인데 객체를 너무 많이 만들게 되면 Stack 저장 공간과 충돌하게 되면서 용량 부족이 나오게 된다.

하지만 Heap에 대해서는 자료구조 시간에 배웠던 기억이 있지만, 아래와 같이 완전 이진트리를 만족하거나 최댓값, 최솟값에 대해서 순서를 지켜 나가는 형식으로 자료가 정리되는 원리를 Heap 자료구조라고 한다.



왼쪽이 최대 Heap, 오른쪽이 최소 Heap이다. Heap이랑 이진트리랑 같다고 생각을 할 수 있겠지만, 실질적으로 차이가 다르다. Heap과 이진 탐색 트리에 대해서 차이점을 알아두면 나중에 피가 되고 약이 되겠다.
Heap >
- 힙은 최댓값, 최솟값에 대해서만 다루기 때문에 이진 트리는 왼쪽 자식, 오른쪽 자식에 대해서 다루는 개념이기보다 레벨(혹은 높이)에 따라서 다루면 된다.
- 힙의 목적은 완전 이진트리인가(혹은 포화 이진트리)에 대해서 만족을 하는 관건으로 두기 때문에 인덱스에 대해서 적용하는 방안이 옳아서 배열로 정의를 하는 경우가 많다.
- 힙은 왼쪽 자식, 오른쪽 자식에 대해서 크게 민감하지 않다. 아래에 있는 37, 27에 대해서는 본래 이진탐색트리에 어긋나지만 여기서는 그런 거 걱정 안 해도 된다. 오로지 작은가, 큰가에 대해서 다루면 된다.

Binary Search Tree>
- 이진 탐색 트리는 루트 노드보다 큰 경우에는 오른쪽으로 보내고, 루트 노드보다 작은 경우에는 왼쪽으로 보내는 원리로 구성되는 자료구조로 볼 수 있는데, 루트 노드가 포화 이진트리인 경우에 중위 순회를 하는 경우 중앙값이 될 수 있다.
- 이진 탐색 트리의 목적은 본래 선형 선택 알고리즘 중에서 이진 탐색 선택 알고리즘을 이용해서 탐색의 속도를 O(log n)으로 줄이는데 있어서 목표를 가지고 있다.
- 하지만 힙과 달리 포화 이진트리(혹은 완전 이진트리)만 완성되는 것이 아니라 계속 작은 값만 적어 나가거나 큰 값만 적어 나가는 경우에 따라서 편향 이진트리가 형성되어서 힙과 달리 시간 복잡도가 O(n)이 되는 최악의 상황이 발생할 수 있다. 또한 이진 탐색 트리는 배열로 표현을 하게 된다면 빈 자리에 대해서 계속 채워나가는 방안밖에 없기 때문에 주로 연결 리스트(Linked List)로 다루는 방안 말곤 없다.

이처럼 힙에 대해서 간략하게 알아봤다. 힙 정렬은 Heap의 삽입/삭제 알고리즘을 통해서 이루어지면 쉽게 할 수 있는 알고리즘이지만, 워낙 복잡한데다가 어려워서 요약정리에는 다루지 않겠다. 이에 대해서는 책을 참조하는 것이 더욱 낫고, 나중에 무글링이나 뭐 시기버에 검색해보면 코드가 나와 있으니 참조하면 좋겠다.